ПРОГРАММИРОВАНИЕ, *2012*, *№ 5*, *с. 1–18*

УДК 004.45

ИСПОЛЬЗОВАНИЕ ДРАЙВЕРОВ УСТРОЙСТВ ОПЕРАЦИОННОЙ СИСТЕМЫ LINUX ДЛЯ СРАВНЕНИЯ ИНСТРУМЕНТОВ СТАТИЧЕСКОЙ ВЕРИФИКАЦИИ *

© 2012 г. М. У. Мандрыкин, В. С. Мутилин, Е. М. Новиков, А. В. Хорошилов, П. Е. Швед

Институт системного программирования РАН

109004 Москва, ул. А.Солженицына, 25

E-mail: mandrykin@ispras.ru, mutilin@ispras.ru, joker@ispras.ru, khoroshilov@ispras.ru, shved@ispras.ru

Поступила в редакцию 08.12.2011

Система верификации Linux Driver Verification предназначена для статического анализа исходного кода драйверов устройств операционной системы Linux. В настоящей статье описывается архитектура системы верификации, включая средства интеграции сторонних инструментов статической верификации Си программ. Рассматриваются характеристики исходного кода драйверов Linux, интересные с точки зрения алгоритмов верификации, приводятся примеры проведения сравнительного анализа различных инструментов верификации, а также различных версий и конфигураций одного инструмента.

1. Введение

Статический анализ кода позволяет проверять выполнимость определенных свойств программ на основе некоторого представления их исходного кода без необходимости реального выполнения программ. Основными преимуществами данного подхода являются то, что во-первых, статический анализ не требует подготовки специального тестового окружения и тестовых данных и может осуществляться сразу после написания исходного кода программы; во-вторых, статический анализ позволяет рассмотреть сразу все пути выполнения программы, в том числе, редко встречающиеся и сложно воспроизводимые при динамическом тестировании.

Практическое применение статического анализа кода имеет некоторые ограничения. Наиболее существенным ограничением при использовании статического анализа кода является время проведения анализа. Дело в том, что современные программы являются очень большими и сложными. В свою очередь это приводит к тому, что выполнение полного статического анализа реальных приложений за разумное время практически невозможно, также как и проведение любого вида исчерпывающего тестирования. Поэтому при практическом применении статического анализа используются различные методы упрощения анализируемых моделей программ и эвристики, которые позволяют получить результат за приемлемое время за счет снижения качества анализа. Ключевыми характеристиками качества статического анализа являются число ложных предупреждений и число пропущенных ошибок искомого вида. В зависимости от степени упрощения и целевого времени работы в рамках статического анализа кода можно условно выделить легковесные и тяжеловесные подходы.

Легковесные подходы нацелены на то, чтобы получать результаты быстро, сравнимо по порядку величины со временем компиляции анализируемого приложения. Для достижения такой высокой скорости данные подходы обычно

*Работа поддержана ФЦП “Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы” (контракт №07.514.11.4104).

используют анализ графа потока данных в сопровождении со множеством различных эвристик, что в конечном итоге отрицательно сказывается на качестве анализа, как в плане количества пропущенных ошибок, так и в плане количества ложных предупреждений. Уменьшение числа ложных предупреждений часто имеет больший приоритет, чем обнаружение всех ошибок, так как опыт использования инструментов статического анализа показывает, что при большом проценте ложных предупреждений общая эффективность их применения значительно падает. Несмотря на это, на сегодняшний день легковесные подходы развиты достаточно хорошо. Существует большое количество различных инструментов, которые их реализуют и широко применяются в промышленной разработке программ. К наиболее успешным коммерческим инструментам относятся Coverity [1] и Klocwork Insight [2], академическим – Svace [3, 4, 5], Saturn [6], FindBugs [7], Splint [8] и др.

При использовании тяжеловесных подходов ограничению по времени работы придается существенно меньшее значение, хотя, тем не менее, время проверки должно оставаться в разумных пределах. Это позволяет использовать значительно меньше эвристик при интерпретации исходного кода программ и, соответственно, применять более качественные методы статического анализа кода, что в свою очередь приводит как к уменьшению числа ложных срабатываний, так и к увеличению числа обнаруживаемых ошибок. На сегодняшний день тяжеловесные подходы мало используются при анализе реальных приложений. Существует большое количество академических проектов, которые предлагают различные реализации тяжеловесных подходов, например, инструменты статической верификации BLAST [9, 10], CPAchecker [11], CBMC [12], ARMC [13] и др. Но в промышленности тяжеловесные методы нашли свое применение только в проекте Microsoft SDV [14], использующем тяжеловесный инструмент статической верификации SLAM [15]. Этот проект следует выделить особо, так как он предоставляет полноценный набор инструментов, позволяющих проводить тяжеловесный статический анализ кода драйверов операционной системы (ОС) Microsoft Windows. Предлагаемые инструменты используются в процессе сертификации драйверов и включены в состав Microsoft Windows Driver Developer Kit, начиная с 2006 года. Проект Microsoft SDV наглядно демонстрирует возможность применения тяжеловесного подхода для верификации реальных программ. Однако, Microsoft SDV является, во-первых, узкоспециализированным, так как по сути нацелен на применение только для драйверов ОС Microsoft Windows, а во-вторых, закрытым, что не позволяет ни расширять область его применения, ни использовать для экспериментов в области алгоритмов статического анализа.

Применение других существующих тяжеловесных инструментов статической верификации на практике носит фрагментарный характер. По сути не существует площадки, на которой можно было бы сравнить характеристики различных инструментов анализа на исходном коде программ, активно используемых в промышленности. В проекте *Linux Driver Verification* [16, 17, 18, 19, 20] сделана попытка построить такую площадку для инструментов (в первую очередь, тяжеловесных), предназначенных для статического анализа программ на языке Си, на примере драйверов устройств ОС Linux. В настоящей статье исследуются требования и предлагается архитектура такой площадки, кроторая позволит как сравнивать различные инструменты верификации между собой, так и сравнивать различные конфигурации одного инструмента верификации.

Настоящая статья построена следующим образом. В разделе 2 сформулированы требования к открытой системе верификации драйверов устройств ОС Linux. В разделе 3 проведен анализ существующих систем статической верификации драйверов для ОС Linux и Microsoft Windows. Раздел 4 содержит подробное описание предложенной архитектуры предложенной системы верификации *Linux Driver Verification*. В разделе 5 обсуждается использование построенной системы верификации для сравнения инструментов статической верификации.

2. Требования к открытой системе верификации

Одной из основных целей проекта *Linux Driver Verification* является построение открытой площадки для экспериментов с различными мето-

дами статического анализа кода, в первую очередь, тяжеловесными, при верификации реальных программ. Для достижения этой цели система верификации должна предоставлять удобные средства для интеграции новых инструментов статической верификации и сравнительного анализа их работы с различными настройками. В качестве целевых программ в проекте рассматриваются драйвера ОС Linux, тем не менее, архитектура инструментария должна предусматривать возможность последующего расширения и на другие приложения.

Драверы ОС Linux являются весьма важными целевыми программами для верификации по следующим причинам:

- Драйверов устройств ОС Linux достаточно много и скорость их появления только возрастает с непрерывно растущей популярностью ОС Linux. Драйверы составляют до 70% исходного кода ядра ОС Linux [21, 22].

- Корректность драйверов является важной составляющей безопасности систем, так как драйверы работают с тем же уровнем привилегий, что и остальное ядро. Более 85% различных ошибок, приводящих к некорректной работе всей ОС, зависаниям и падениям находятся именно в драйверах [21, 22, 23]

Обеспечивать надежность драйверов Linux вручную, даже несмотря на большое количество разработчиков (более 1000 человек на сегодняшний день [24]), весьма затруднительно ввиду огромного количества достаточно сложного исходного кода (более 13 млн строк кода [24]), который должен удовлетворять достаточно большому числу разнообразных правил корректности, начиная от общих правил, которым должны подчиняться все программы на Си, и заканчивая специфичными правилами, которые говорят о том, как драйверы должны использовать интерфейс ядра.

Драйвера ОС Linux имеют ряд преимуществ с точки зрения применения статического анализа:

- Большинство драйверов публикуются вместе с исходным кодом, который является необходимым для большинства инструментов статического анализа.

- Драйверы не используют арифметику с плавающей точкой, редко используют рекурсивные функции.

- Драйверы достаточно небольшие по размеру, а потому можно предположить, что время проверки одного драйвера будет сравнительно невелико.

Для успешного применения системы верификации к драйверам устройств ядра ОС Linux требуется, чтобы система была готова к промышленному применению. Это означает, что должно быть минимизированно участие человека в настройке инструментов и предоставлен максимально удобный интерфейс для использования системы. С точки зрения минимизации участия человека, в первую очередь, необходимо автоматизировать извлечение информации о составе драйвера и настройках его компиляции из уже имеющихся данных, предназначенных для сборки драйвера. При этом важно учитывать, что с одной стороны у драйверов есть много различных зависимостей, а с другой – что для эффективного применения статического анализа количество строк кода должно быть не очень большим.

Другая потребность в автоматизации связана с отсутствием традиционной точки входа (иными словами, функции *main*) у драйверов устройств. Для большинства тяжеловесных подходов статического анализа кода наличие точки входа является необходимым условием в виду того, что они исследуют пути выполнения в программе, начиная от данной точки. Поэтому для проведения верификации драйверов требуется генерация модельного окружения драйвера в виде искусственной точки входа, где должны вызываться функции обработчики драйверов (например, функция инициализации драйвера, чтения с устройства и т.п.) на манер того, как это делается при реальном взаимодействии драйверов, ядра и оборудования.

Важно, чтобы систему верификации было удобно использовать, что включает в себя удобство запуска верификации и удобство анализа ее результатов. Причем последнее является наиболее значимым, так как анализ выявленных ошибок может занимать немалое время и требовать привлечения высококвалифицированных,

а значит и дорогостоящих, специалистов.

3. Существующие решения

В данном разделе рассмотрены существующие системы верификации драйверов, основывающиеся на тяжеловесных методах статического анализа кода, в первую очередь, с точки зрения сформулированных требований к построению открытой системы верификации, подходящей для промышленного использования.

Наиболее полноценным образом подход реализуется в уже упоминавшемся ранее системе Microsoft SDV [14]. Данная система предоставляет широкие возможности по верификации драйверов ОС Microsoft Windows с помощью статического анализа драйверов, входящих в состав ОС Windows, так и в существующей программе сертификации драйверов сторонних разработчиков. К особенностям подхода относятся следующие:

- Для создания окружения от пользователя требуется вручную аннотировать исходный код драйверов, указывая в нем роли каждой из функций обработчиков.

- Проверяемые правила корректности формализуются с помощью языка SLIC [25], в котором связь с исходным кодом драйвера задается с помощью аспектно-ориентированных конструкций, перехватывающих вызовы функций ядра. В настоящее время уже выделен набор из более чем 210 правил, а в исследовательской версии была реализована возможность добавления новых правил. Следует отметить, что в отличие от ядра Linux в ядре Microsoft Windows интерфейс меняется гораздо реже, поэтому проблема подстраивания под изменения ядра для Microsoft SDV не столь актуальна.

- Известно, что в собственных исследовательских целях разработчики Microsoft SDV могут подключать два инструмента статической верификации SLAM и Yogi [26]. Подключение инструментов верификации сторонними разработчиками не предусмотрено.

- Имеется возможность просмотреть сводную статистику по всем проверяемым правилам для анализируемого драйвера. Для найденных ошибок можно просмотреть представленные удобным и наглядным образом трассы ошибок, которые связаны с соответствующим исходным кодом драйверов и ядра. Кроме того, при анализе ошибок, пользователю показываются подсказки.

Существует также несколько систем, использующих тяжеловесные методы статического анализа кода для верификации драйверов ядра ОС Linux: DDVerify [27], разработка университета Карнеги-Меллон (США) и Avinux [28], разработка университета города Тюбинген (Германия).

Особенности системы DDVerify таковы:

- Информацию о составе и настройках сборки драйверов DDVerify получает, используя собственные файлы сборки без учета файлов сборки ядра. Поэтому система не учитывает специфику компиляции ядра в полной мере.

- Для создания окружения используется модель ядра для некоторых типов драйверов. Разработчиками были написаны модели для трех типов драйверов, а всего их несколько десятков.

- Правила корректности задаются как часть модели ядра. Код ограничений, накладываемых правилом, задается вместе с кодом, описывающим семантику функции. В данном подходе можно проверять только те функции, которые привязываются к драйверу с помощью линковки. Поэтому для использования DDVerify требуется существенно изменять заголовочные файлы ядра.

- Система позволяет подключать два инструмента статической верификации: CBMC [12] и SATABS [29].

- Для анализа результатов анализа имеется специальный плагин для интегрированной среды разработки Eclipse. В частности, имеется возможность анализа трассы ошибки с одновременным просмотром соответствующего исходного кода драйверов и ядра ОС Linux, что существенно облегчает анализ трасс ошибок.

Система Avinux, которая также предназначена для верификации драйверов ядра ОС Linux, обладает следующими характерными особенностями:

- Получение исходного кода драйвера для последующей верификации происходит на основе встраивания в процесс сборки ядра путем модификации файлов, описывающих сборку. Однако, Avinux предоставляет возможность автоматической работы только с единичными препроцессированными файлами. Поэтому, например, верификация драйверов, состоящих из нескольких файлов возможна только вручную, так как информация о зависимостях теряется.

- Для создания окружения драйвера, требуется вручную написать функцию *main*, эмулирующую использование драйвера ядром операционной системы и, кроме того, выбрать файлы, входящие в драйвер. На основе этого код инициализации параметров генерируется автоматически [30].

- Для задания правил используются аспектно-ориентированные конструкции похожие на конструкции SLIC.

- Система интегрирована с единственным инструментой статической верификации CBMC [12].

- Для упрощения запуска Avinux предоставляет плагин к среде Eclipse, но средства для визуализации трассы ошибок отсутствуют.

Таким образом, ни одина из рассмотренных систем не позволяет полностью достигнуть поставленных целей. Система Microsoft SDV является закрытой и предназначена только для верификации драйверов ОС Microsoft Windows. Системы Avinux и DDVerify не подходят для широкомасштабного использования. Avinux требует описания функций обработчиков драйвера и не поддерживает драйверы, состоящие из нескольких файлов. DDVerify требует серьезной переработки собственного процесса сборки, заголовочных файлов и модели ядра для каждой новой версии ядра, что заметно усложняет сопровождение системы в условиях большого количества изменений в ядре ОС Linux.

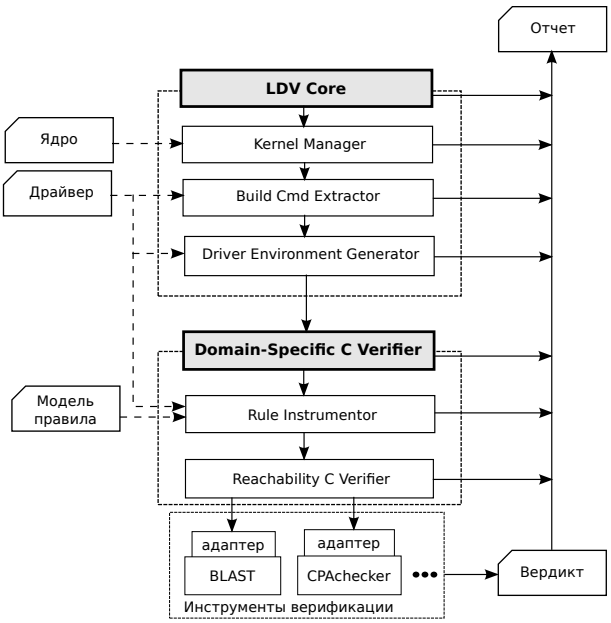


Рис. 1.: Архитектура системы верификации *LDV*

В случае обнаружения ошибки некоторые существующие системы предоставляют пользователю возможности анализа трассы ошибки, приводящей к обнаруженному нарушению правила корректности, но ни один из рассмотренных систем не поддерживает интеграцию сторонних инструментов верификации, а также сравнительный анализ результатов. В табл. 1 приведена сравнительная информация по рассмотренным системам с точки зрения выявленных требований к открытой системе верификации.

4. Архитектура системы верификации драйверов

Архитектура системы верификации *Linux Driver Verification (LDV)* разрабатывалась для достижения описанных ранее целей: предоставить высокоавтоматизированную инфраструктуру для проверки драйверов ядра ОС Linux и возможность подключения различных инструментов, реализующих тяжеловесные подходы статического анализа кода.

Схематично архитектура *LDV* изображена на рис. 1. Основные компоненты архитектуры изображены в центре в порядке вызова. Слева изображены входные данные, которые предоставляются пользователем. Стрелки показывают то, как компоненты архитектуры используют вход-

Требования	**Microsoft SDV**	**Avinux**	**DDVerify**
Переиспользование информации о параметрах сборки	+	±	–
Генерация модели окружения	По аннотации	Ручная	Для 3-х типов драйверов
Поддержка добавления новых правил корректности	+	+	±
Сопровождаемость в условиях непрерывного развития ядра	Не требуется	±	–
Визуализация результата и трассы ошибки	+	–	±
Поддержка интеграции новых инструментов верификации	–	–	–
Поддержка сравнительного анализа результатов	–	–	–

Таблица 1.: Сравнение существующих систем верификации драйверов

ные данные. Справа показан порядок формирования отчета о результатах верификации.

Система верификации *LDV* предоставляет возможность интеграции сторонних инструментов статической верификации посредством реализации адаптеров. В настоящее время система включает адаптеры для двух инструментов статической верификации, BLAST и CPAchecker.

Далее приведено описание компонентов и пользовательского интерфейса системы *LDV*.

4.1. LDV Core

Процесс верификации драйверов начинается с запуска компонента *LDV Core*. Данный компонент вызывает компонент *Kernel Manager*, который создает на диске копию ядра ОС Linux, предоставленного пользователем в виде архива с исходным кодом или репозитория git. Затем *Kernel Manager* модифицирует подсистему сборки ядра, что впоследствии позволяет получить информацию о составе драйверов и настройках их сборки. Пользователь может верифицировать как драйверы, входящие в предоставляемое им ядро ОС Linux, так и внешние драйверы относительно данного ядра.

Далее *LDV Core* запускает процесс компиляции копии ядра, по ходу которого на основе модифицированной подсистемы сборки *Build Cmd Extractor* читает поток команд компиляции и линковки, определяет зависимости между файлами с исходным кодом и выделяет то, что относится к верифицируемым драйверам.

После того, как получен полный поток команд, специальный подкомпонент *Build Cmd Extractor* разделяет данный поток на несколько небольших частей, каждая из которых соответствует одному модулю ядра. Эта процедура особенно важна для анализа драйверов, входящих в ядро, поскольку сама по себе подсистема сборки ядра не делает такого разделения.

Драйвер ядра ОС Linux, как правило, состоит из одного или нескольких модулей. Эти модули не могут быть исполнены непосредственно как программы – вместо этого они предоставляют ядру функции обработчики событий, такие как операции с файлами, операции с USB интерфейсом и т.п.; а также обработчики прерываний, таймеры, обработчики отложенных задач. Функции обработчики регистрируются при загрузке драйвера в ядро, а затем вызываются по мере поступления соответствующих запросов со стороны пользовательских приложений и оборудования.

Основная задача компонента *Driver Environment Generator*, который запускает *LDV Core*, состоит в том, чтобы сгенерировать одну или несколько моделей окружения для каждого типа драйвера. Модели окружения

включают в себя загрузку и выгрузку драйвера, а также вызовы функций обработчиков наподобие того, как это происходит при реальном взаимодействии пользовательских приложений и оборудования с ядром ОС Linux. Различия моделей заключаются в том, что функции обработчики в них вызываются по-разному, например, обработчики прерываний могут вызываться или нет. Сгенерированные модели печатаются в виде функций псевдо *main* на языке программирования Си. Впоследствии данные функции служат точками входа для инструментов статической верификации, что будет отражено в следующем подразделе данной статьи.

На данный момент *Driver Environment Generator* позволяет генерировать следующие модели окружения:

1. Фиксированная последовательность вызовов функций обработчиков.
2. Произвольная последовательность вызовов функций обработчиков ограниченной длины.
3. Произвольная последовательность вызовов функций обработчиков неограниченной длины.

Помимо этого, в модели окружения объявляются переменные, которые используются в качестве аргументов функций обработчиков. Инициализация данных переменных при построении модели окружения не поддерживается, однако, это может быть сделано, например, с помощью внешнего инструмента такого, как DEC[30].

При построении модели окружения соблюдаются ограничения на порядок вызовов функций обработчиков (предусловия вызовов). Например, учитывается, что в корректной последовательности вызов функции *read* может встречаться только после успешного вызова функции *open*.

4.2. *Domain Specific C Verifier*

Domain Specific C Verifier предоставляет интерфейс для статической верификации программ на языке Си, не зависящий от способа описания моделей правил и от используемого инструмента статической верификации. На вход *Domain Specific C Verifier* поступает набор команд сборки, соответствующих анализируемому модулю, наборы точек входа и идентификаторов моделей правил, которые необходимо проверить.

Для каждого идентификатора модели правила *Domain Specific C Verifier* вызывает *Rule Instrumentor*, который создает задание для соответствующего инструмента статической верификации. Взаимодействие с инструментом верификации осуществляет *Reachibility C Verifier* посредством специального адаптера, специфичного для каждого инструмента статической верификации.

Отметим, что информация о верификации, специфичная для драйверов и ядра ОС Linux, полностью скрывается в описании модели правила, поэтому компонент *Domain Specific C Verifier*, вообще говоря, может быть применен и для других предметных областей в том случае, если предоставлены соответствующие описания моделей правил корректности, набор команд сборки и исходный код анализируемой программы. Для примера, уже на сегодняшний день в базе моделей правил описано общее правило, проверяющее, что в программе не нарушается ни один *assert*. Данное правило может быть применимо для произвольной программы на языке Си.

Rule Instrumentor – компонент, основное назначение которого – связывать формализованное представление моделей правил корректности с исходным кодом проверяемой программы для его последующей верификации с помощью некоторого инструмента статической верификации.

Помимо исходного кода программы и команд сборки на вход *Rule Instrumentor* поступает идентификатор модели правила. Информация о моделях хранится в базе данных моделей. Используя идентификатор, *Rule Instrumentor* получает описание соответствующей модели, которое, по сути, состоит из путей к так называемым *аспектным файлам* и вспомогательной информации для инструмента верификации, которая способствует выполнению более качественной и быстрой проверки.

Аспектные файлы пишутся на аспектно-ориентированном расширении языка программирования Си наподобие того, как это делается с помощью SLIC в проектах SDV и Avinux.

Данный подход, с одной стороны, предоставляет достаточно большой набор средств для моделирования ошибок, причем при этом верифицируемые свойства можно задавать с помощью интуитивно понятного языка. С другой стороны, подход позволяет описывать модели ошибки, независимым от используемого инструмента статической верификации образом, что важно ввиду нацеленности *LDV* на различные инструменты статической верификации.

В качестве примера далее приведены упрощенное формализованное представление правила корректности блокировки мьютексов и общая схема инструментирования исходного кода. Мьютексы позволяют монопольно выполнять критические секции программ. Однако, данные объекты следует использовать осторожно, поскольку, например, попытка повторной блокировки одного и того же мьютекса может привести к зависанию программы. Подобных ситуаций не должно возникать в программе, поэтому правило говорит, что один и тот же мьютекс не должен блокироваться дважды одним и тем же процессом.

Для формализации правила корректности блокировки мьютексов в *Аспекте 1* определяются глобальная переменная *islocked*, которая хранит текущее состояние мьютекса, и модельная функция *model_mutex_lock*. Данная функция проверяет состояние мьютекса. В том случае, если он уже был заблокирован, метка *ERROR* сообщает о нарушении правила. В противном случае изменяется текущее состояние мьютекса. В *Аспекте 1* опущено определение модельной функции *model_mutex_unlock*, которая похожа на *model_mutex_lock*.

Аспект 1 – модельные состояние и функции:

```
// Инициализация модельного состояния
int islocked = UNLOCKED;
// Определение модельной функции
void model_mutex_lock() {
 // Проверка правила
 if (islocked == LOCKED)
  // Ошибочная точка
  ERROR: abort();
 // Моделирование поведения функции
 islocked = LOCKED;
}
```

Аспект 2 задает точки в исходном коде драйвера, которые должны быть дополнены вызовами модельных функций, определенных в *Аспекте 1*. Для упрощения в *Аспекте 2* опущена связь с модельной функции *model_mutex_unlock*.

Аспект 2 – связь конструкций исходного кода с Аспектом 1:

```
// Перед вызовом mutex_lock
before: call($ mutex_lock(..)) {
 // вызвать модельную функцию
 model_mutex_lock();
}
```

В результате инструментирования исходного кода драйвера, которое выполняется на основе *Аспекта 1* и *Аспекта 2*:

```
...
// Вызов функции mutex_lock
mutex_lock(&socket_mutex);
...
```

получается следующий исходный код, где перед вызовом исходной функции блокировки мьютекса вызывается модельная функция (опущены определения модельного состояния и модельной функции):

```
...
// Вызов вспомогательной функции
ldv_mutex_lock(&socket_mutex);
...
// Определение вспомогательной функции
void ldv_mutex_lock(struct mutex *arg) {
 // Вызов модельной функции
 model_mutex_lock();
 // Вызов функции mutex_lock
 mutex_lock(arg);
}
```

На сегодняшний день в проекте *Linux Driver Verification* для анализа инструментированного исходного кода драйверов ОС Linux используются так называемые *инструментами верификации достижимости*, то есть инструменты статической верификации, предназначенные для выявления нарушений правил корректности, выраженных в виде достижимости ошибочной точки в программе. Тем не менее, в дальнейшем планируется реализовать поддержку и других классов инструментов верификации, например, решающих задачу завершаемости, что не потребует внесения значительных изменений в архитектуру системы верификации.

Инструменты верификации достижимости вы-

носят один из трех вердиктов: SAFE, UNSAFE и UNKNOWN. Вердикт SAFE означает, что соответствующий инструмент статической верификации гарантирует отсутствие нарушений проверяемого правила корректности. Вердикт UNSAFE говорит об обнаружении нарушения правила и сопровождается более детальной информацией о проблеме. В случае инструментов верификации достижимости, такой информацией, как правило, является трасса ошибки, которая показывает путь выполнения программы, который приводит к ошибочному состоянию. Вердикт UNKNOWN означает, что инструмент по тем или иным причинам (например, нехватка памяти или времени) не смог найти однозначного ответа на поставленный вопрос.

Компонент *Reachibility C Verifier* решает задачу преобразования задачи верификации из представления *LDV* в виде набора команд сборки и настроек верификации в представление конкретного инструмента верификации. Данный компонент получает на вход инструментированный исходный код драйверов со сгенерированными точками входа и ошибочными метками. На данном исходном коде *Reachibility C Verifier* вызывает заданный адаптер для соответствующего инструмента верификации достижимости.

В типовом адаптере инструмента статической верификации можно выделить следующие четыре части:

- подготовка входных файлов;
- подготовка обработчика вывода инструмента верификации;
- запуск инструмента;
- обработка результатов.

Далее в статье рассмотрены данные части по отдельности с указанием того, какие средства предоставляются разработчику адаптера для реализации соответствующих задач.

В рамках одной задачи инструменту верификации необходимо проверить достижимость ошибочной метки из некоторой точки входа, причем анализируемая программа может быть расположена в одном или нескольких файлах. Инструменты верификации могут накладывать ограничения на то, в каком виде должны быть представлены эти файлы. Разработчику адаптера предоставляется возможность применить к каждому из входных файлов стандартный препроцессор языка Си, обработать файлы с помощью инструмента трансформации кода CIL, или соединить все файлы в один с помощью того же CIL [31]. В результате возвращается список файлов, в которых содержится весь необходимый для проверки исходный код.

Обработчику вывода инструмента верификации на вход подаются строчки, выдаваемые инструментом верификации на стандартный вывод и/или на стандартный поток ошибок. Обработчик опционально возвращает набор значений с некоторой информацией, которую он извлек из трассы (например, последние 20 строк или вердикт о наличии/отсутствии в программе ошибок). Каждая функция обработчик хранит свое внутреннее состояние. Непосредственно перед запуском инструмента верификации адаптер регистрирует такие функции, а *LDV* применяет их, когда инструмент верификации будет запущен, параллельно с его работой.

Запуск инструмента верификации заключается в вызове библиотечной функции с командной строкой, соответствующей вызову инструмента верификации. Подготовка аргументов осуществляется в индивидуальной для каждого инструмента манере, на основе полученных имен препроцессированных файлов, точки входа и ошибочной метки. Библиотечная функция, через которую адаптер осуществляет вызов инструмента, отличается от стандартной функции вызова внешней программы некоторыми функциональными особенностями, а именно:

- автоматически применяет функции обработчики вывода, зарегистрированные адаптером ранее;
- вывод инструмента, который может быть большим по объему, архивируется и сохраняется на диск;
- инструмент запускается в контролируемом окружении, позволяя лимитировать использование инструментом и его дочерними процессами ресурсов машины, а именно потребления памяти, диска и процессорного времени;

- по желанию разработчика адаптера, происходит измерение потребления запускаемым инструментом и всеми его дочерними процессами времени; при этом разработчик адаптера может указать, измерение времени в каких именно процессах ему интересно и в какие категории каждый из них следует отнести.

После окончания запуска адаптер получает информацию о причине окончания (нарушение лимита ресурсов, получение сигнала или успешное завершение), коде возврата и номеру завершившего процесс сигнала, а также информацию, собранную функциями обработчиками вывода трассы. Ожидается, что в адаптере разработчик реализует автоматизированную интерпретацию этой информации. Например, разработчик адаптера должен реализовать интерпретацию трассы ошибки соответствующего инструмента статической верификации и ее преобразование в специальный общий формат, используемый *LDV* для визуализации этой трассы. Подробней общий формат трассы описан в подразделе, описывающем пользовательский инстерфейс. Адаптер также может передать произвольную текстовую строку (например, содержащую исключение и трассу стека, выброшенные инструментом при неудачном завершении) и один или несколько файлов для сохранения их в финальном отчете с результатами верификации. Затем эта информация может быть использована для последующей обработки и построения статистики с помощью компонента *LDV Statistics Server*, который рассмотрен в разделе про пользовательский интерфейс.

Разработчик адаптера также может добавить конфигурируемость адаптеров, чтобы проводить эксперименты с различными настройками инструментов верификации без необходимости модифицировать код адаптера каждый раз. Например, для конфигурации разработчик адаптера может использовать переменные окружения.

На сегодняшний день предложенный интерфейс интеграции со сторонними инструментами верификации был опробован на примере инструментов статической верификации BLAST и CPAchecker. Было выявлено, что предоставленного интерфейса достаточно, чтобы обеспечить нужды этих инструментов по интеграции в систему верификации *LDV*.

4.3. Обработка результатов анализа

Вердикт о результате верификации и другая информация, выдаваемая инструментом статической верификации, обрабатывается всеми компонентами на рис. 1 в обратном порядке. При этом каждый компонент дополняет отчет информацией о своей работе, после чего формируется финальный отчет о проверке всего задания.

Компоненты между собой общаются с помощью потока команд (в настоящее время он представляется в виде XML). Изначально поток команд представляет команды сборки драйверов, но по мере работы он может модифицироваться каждым компонентом. Компоненты могут изменять опции препроцессора, дописывать метаинформацию и даже подменять пути к файлам теми, по которым они располагают модифицированные файлы. Все компоненты имеют четкий и документированный интерфейс, а потому могут быть доработаны или подменены независимым образом.

4.4. Пользовательский интерфейс системы

Пользователь взаимодействует с системой верификации *LDV* посредством высокоуровневого интерфейса командной строки *LDV manager*. Данный компонент позволяет проверить некоторый набор драйверов (внутренних или внешних) для некоторого набора ядер по одному или нескольким правилам корректности. В том случае, когда в ходе работы не происходит критическая исключительная ситуация, на выходе *LDV manager* создает архив, содержащий результаты анализа, информацию о работе компонентов архитектуры *LDV*, трассы ошибок и необходимые для их визуализации файлы с исходным кодом драйверов и ядра ОС Linux. Далее данный архив может быть загружен в базу данных и использован для анализа, например, с помощью *Statistics Server*.

Statistics Server – это компонент, который предоставляет веб-интерфейс для статистического анализа и сравнения результатов верификации.

Statistics Server позволяет анализировать большие объемы данных, получаемых в ходе

верификации драйверов различных версий ядра ОС Linux по множеству правил с помощью инструментов статической верификации, запускаемых с разными конфигурациями. Помимо статистки, такой как, например, суммарное количество различных вердиктов для некоторого ядра и правила, компонент позволяет анализировать детальные списки, например, посмотреть все драйверы некоторого ядра, для которых инструмент верификации выдал вердикт UNSAFE на некотором правиле корректности.

Система верификации *LDV* изначально затачивалась для использования различной целевой аудиторией такой, как разработчики компонентов, разработчики ядра, разработчики инструментов верификации достижимости и т.д. Как правило, запросы к представлению статистики у этих групп отличаются, поэтому *Statistics Server* предлагает различные заранее подготовленные профили представления данных. Так, например, разработчикам инструментов верификации достижимости помимо статистики по вердиктам предоставляется статистика по времени, затраченном на верификацию; разработчикам компонентов показывается статистика по внутренним проблемам соответствующих компонентов.

Еще одна важная возможность компонента – это возможность сравнения результатов различных заданий верификации. В частности, это оказывается очень полезным и удобным инструментом для сравнения различных инструментов статической верификации, в том числе, различных версий и конфигураций. Более подробное описание данной возможности рассмотрено в соответствующем разделе данной статьи.

Statistics Server интегрирован с *Error Trace Visualizer*, компонентом, который нацелен на упрощение анализа трасс ошибок, которые выдают инструменты верификации достижимости в случае вынесения вердикта UNSAFE. Данный компонент позволяет ускорить анализ трасс ошибок, благодаря чему существенно повышается степень автоматизации процесса верификации в целом. По сравнению с работой [32], где *Error Trace Visualizer* описан достаточно подробно, в данной статье дается краткая характеристика компонента, а также описываются последние разработки данного компонента.

Как правило, трасса ошибки представляется в текстовом виде, который имеет весьма специфичный, вообще говоря, сильно зависящий от инструмента верификации, формат. Для некоторых инструментов статической верификации существуют инструменты, позволяющие представить трассы ошибок в более наглядном и удобном для анализа формате. Например, трассу ошибки инструмента статической верификации CPAchecker можно преобразовать в HTML, после чего ее можно открывать в любом браузере. Трассы ошибок SATABS визуализируются посредством специального Eclipse плагина. А, например, инструмент статической верификации BLAST до *Error Trace Visualizer* не имел инструментов, позволяющих упростить анализ трасс ошибок. Подобное многообразие форматов представления трасс ошибок, в конечном итоге, затрудняет их анализ для различных инструментов верификации.

В рамках проекта *LDV* был разработан общий формат представления трасс ошибок. Разработанный формат является в достаточной степени гибким и расширяемым, что позволяет преобразовывать к нему трассы ошибок различных инструментах верификации без больших трудозатрат. Преобразование исходных трасс ошибок к общему формату реализуется на уровне адаптеров инструментов верификации. Для инструментов BLAST и CPAchecker подобное преобразование было реализовано разработчиками *LDV*. *Error Trace Visualizer* визуализирует трассы, представленные в общем формате, единообразным образом и показывает результаты с помощью веб-интерфейса.

Важно отметить, что при визуализации наряду с трассой ошибок *Error Trace Visualizer* показывает соответствующий ей исходный код программы, причем, между ними устанавливаются определенные взаимосвязи (например, соответствие строк трассы ошибки строкам исходного кода программы). Также компонент выделяет каждый класс элементов трассы ошибки определенным стилем и цветом. Для показываемого исходного кода программы выполняется синтаксическая подсветка. Имеется возможность скрывать и раскрывать как отдельные элементы, так и целые классы элементов трассы ошибки. Все это существенно облегчает анализ трасс ошибок различных инструментов верификации.

Следует отметить тот факт, что хотя в настоящее время компоненты *Statistics Server* и *Error Trace Visualizer* заточены на визуализацию результатов верификации для драйверов ОС Linux, они могут быть адаптированы для произвольных программ.

4.5. Резюме

В данном разделе была представлена система верификации драйверов устройств *LDV*, которая поддерживает встраивание внешних инструментов статической верификации с помощью написания адаптеров. Система интегрирована с процессом сборки ядра, поэтому вся необходимая информация извлекается при сборке. Генерация окружения осуществляется полностью автоматически, поэтому можно с легкостью верифицировать любое количество драйверов. Система позволяет добавлять новые правила корректности с помощью аспектно-ориентированного расширения языка программирования Си. Для сопровождения в условиях непрерывного развития ядра используется набор регрессионных тестов, которые обнаруживают места, требующие изменений, однако, изменение аспектов, связывающих модельные функции с исходным кодом ядра производится вручную. Для анализа трасс ошибок и сравнительного анализа результатов реализованы специальные компоненты.

5. Сравнение инструментов статической верификации

Сравнение инструментов верификации предполагает наличие эталонного набора данных, т.е. постоянного и представительного множества данных, которые можно использовать для сравнения. Существуют подходы к подготовке наборов, в которых в качестве эталонных выбираются коллекции небольших программ, нацеленных на конкретные сложности возникающие при верификации, такие как анализ указателей, побочные эффекты, передача управления, и т.д. [33]. Подход, предлагаемый в данной работе, состоит в использовании для сравнения рельного промышленного исходного кода, которым являются драйверы устройств ОС Linux. Для реализации подхода система верификации *LDV* предоставляет возможность подключения новых инструментов статической верификации, а также возможность сравнительного анализа результатов верификации. Кроме того, система *LDV* допускает верификацию программ не являющихся драйверами, поэтому программы, сконструированные специально для проверки проблемных мест инструмента верификации, также могут быть включены в результирующий набор.

5.1. Характеристики драйверов устройств ОС Linux

Для демонстрации типичных характеристик Си программ, подготавливаемых системой *LDV* для верификации, рассмотрим пример применения правила, описывающего использование захвата и освобождения мьютексов, к драйверам, поставляемым как модули ядра ОС Linux версии 2.6.31.6. Таковых насчитывается 2158 драйвера, но только 712 из них используют мьютексы. Ниже представлены только последние, проанализированные с использованием окружения: потенциально бесконечные последовательности вызовов обработчиков в произвольном порядке.

Исходный код драйверов был проанализирован инструментом статической верификации BLAST [10]. Данный инструмент сохранял информацию о том, сколько он выполнил уточнений абстракции, сколько предикатов было выделено при анализе трасс ошибок интерполирующей процедурой (при анализе одной трассы может быть выделено несколько предикатов). Были выключены все эвристики, которые могли бы уменьшить количество итераций уточнения осуществляемых в традиционном анализе CEGAR [34]. Запуски BLAST принудительно завершались по истечении 10 минут и, таким образом, полученные результаты являются оценкой снизу реальных характеристик драйверов.

Характеристики драйверов показаны в таблице 2. Драйверы разделены на группы по количеству строк в .c файлах. Для всех драйверов каждой группы представлены следующие характеристики: общее количество запусков инструмента верификации (N); размер графа потока управления (ГПУ), в частности количество ребер в нем; средняя и максимальная длина трассы ошибки для каждого драйвера, где длина измеряется как количество элементов в конъюнктивной форме формулы пути; количество уточнений

Строк	N	Ребер ГПУ	Длина трассы		Уточнения		Предикаты	
от – до		среднее	ср.	макс.	ср.	макс.	ср.	макс.
0 – 1999	340	1356	853	6102	19	224	14	123
2000 – 3999	129	3073	1750	10997	36	372	26	215
4000 – 5999	57	4855	1993	7326	18	226	15	95
6000 – 7999	61	7396	5873	30531	16	95	17	143
8000 – 9999	29	8349	4708	13518	29	99	16	67
10000–11999	35	14038	4161	8082	10	93	10	62
12000–13999	11	10588	8365	12103	10	58	6	23
14000–25208	50	18054	9396	18949	6	47	6	75

Таблица 2.: Характеристики драйверов, подготавливаемых системой *LDV* для инструментов статической верификации (драйверы сгруппированы по размеру)

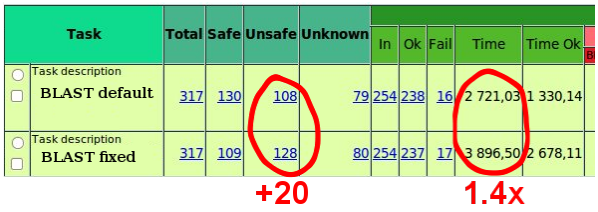


Рис. 2.: Результаты запуска инструмента BLAST до (*default*) и после (*fixed*) исправления ошибки

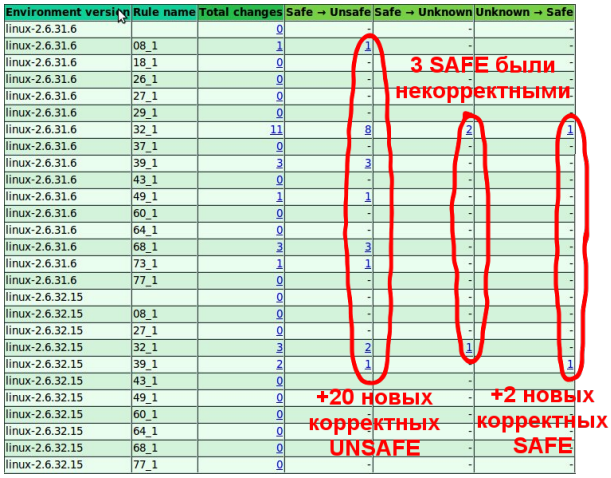


Рис. 3.: Изменения вердиктов после исправления ошибки в инструменте BLAST (показаны изменения *default* → *fixed*)

абстракции (каждое уточнение непосредственно следует за анализом трассы ошибки); количество предикатов, добавленных в абстракции до завершения работы инструмента верификации.

Приведенные в таблице 2 данные демонстрируют, что большинство драйверов имеют размер менее чем 12 тысяч строк кода. Количество уточнений в среднем невелико, но существуют драйверы, дающие хорошую нагрузку для алгоритмов CEGAR и требующие существенного количества итераций уточнения для относительно простого правила корректности.

Далее рассмотрены возможности системы верификации *LDV* в части использования для разработки инструментов верификации, сравнения результатов работы одного инструмента в различных конфигурациях и сравнения нескольких инструментов верификации.

5.2. *Сравнение версий одного инструмента*

В процессе анализа результатов работы инструмента верификации BLAST было обнаружено, что подсистема инструмента, отвечающая за адаптивный статический анализ (Configurable Program Analysis, CPA)[35], содержала ошибки, приводившие к пропуску ошибок. Разработчики *LDV* исправили данные ошибки, о чем написано в работе [10]. На рис. 2 можно видеть статистику по двум запускам. Первый запуск *BLAST default* – это запуск до исправлений, *BLAST fixed* – запуск после исправлений. Можно видеть, что время работы инструмента (в секундах) увеличилось не столь существенно, с другой стороны инструмент стал обнаруживать пропущенные ранее UNSAFE.

При детальном сравнении первого запуска со вторым (рис. 3) можно видеть, что во втором

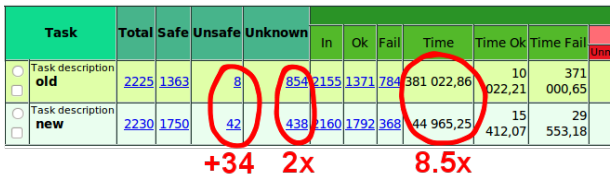


Рис. 4.: Сравнение инструмента BLAST версий 2.5 (*old*) и 2.6 (*new*)

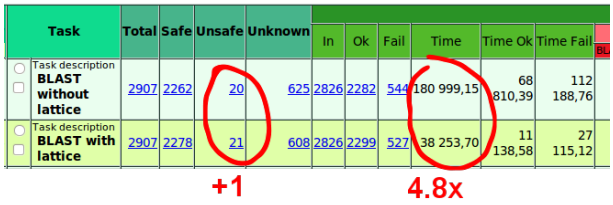


Рис. 6.: Сравнение конфигураций инструмента BLAST *without lattice* и *with lattice*

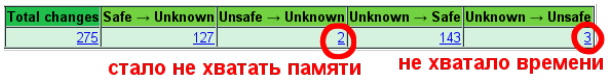


Рис. 7.: Изменения вердиктов для запуска BLAST в различных конфигурациях (показаны изменения *without lattice* → *with lattice*)

запуске было найдено 20 новых корректных UNSAFE драйверов, которые в первом были SAFE. Было найдено 2 новых SAFE, которые были UNKNOWN, 3 драйвера, которые в первом запуске были SAFE (в результате ошибки в инструменте BLAST), а стали UNKNOWN.

На рис. 4 представлены результаты сравнения инструмента BLAST версии 2.5, выпущенной в 2008 году (*old*) с версией 2.6, выпущенной в 2011 году (*new*). Отчет демонстрирует, что существенно увеличилась скорость работы инструмента. Вместо 105 часов на одном правиле и всех драйверах ядра Linux, BLAST стал работать 12,5 часов, что в 8.5 раз быстрее. Кроме того, новая версия дает в два раза меньше вердиктов UNKNOWN и находит 34 новых UNSAFE.

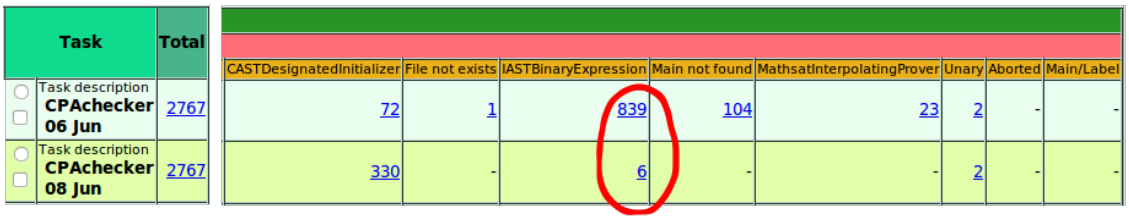
Система *LDV* позволяет идентифицировать наиболее критичные ошибки, которые препятствуют анализу наибольшего количества драйверов. Для этого используется шаблоны ошибок, которые применяются к выводу сообщений об ошибках инструментов и, таким образом, ошибки объединяются в группы в результирующем отчете. Например, после запуска инструмента CPAchecker и добавления шаблонов ошибок было выявлено, что наибольшее их количество связано с исключением IASTbinaryExpression (рис. 5). Об этой ошибке было сообщено разработчикам инструмента, при этом было указано, что ошибка является очень критичной, так как инструмент CPAchecker падает на 30% всех драйверов. В результате, ошибка была исправлена в течении одного дня, и количество драйверов с данной ошибкой уменьшилось до приемлемого (менее 0,2%).

5.3. *Сравнение конфигураций одного инструмента*

Система верификации *LDV* также может быть использована для сравнения различных конфигураций одного инструмента и оценки эффективности в применении к драйверам Linux. Например, в инструменте BLAST есть опция *-lattice*, включающая адаптивный статический анализ. Запуски инструмента без использования данной опции (*BLAST without lattice*) и с ее использованием (*BLAST with lattice*) показаны на рис. 6. Видно, что включение адаптивного статического анализа дает выигрыш по скорости в 4.8 раз при сравнимом количестве UNSAFE, 20 и 21 соответственно. Детальный анализ (рис. 7) показывает, что на самом деле различий по UNSAFE больше, чем в одном драйвере: два UNSAFE стали UNKNOWN по причине недостатка памяти, три UNKNOWN, вызванных превышением лимита времени, стали UNSAFE. Таким образом, включение опции *-lattice* требует больше памяти, тогда как без нее требуется больше времени.

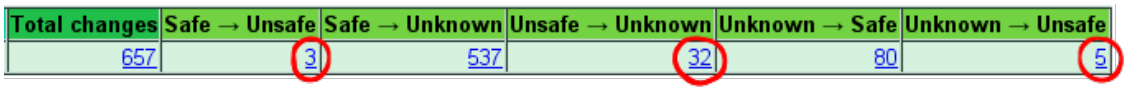
5.4. *Сравнение разных инструментов*

Система *LDV* может быть использована для сравнения нескольких инструментов статической верификации. Например, было произведено сравнение инструментов BLAST 2.6 и CPAchecker (версия от 10 июня 2011) в применении к драйверам. Сводная статистика показала, что CPAchecker работает примерно в три раза



Task	Total	CASTDesignatedInitializer	File not exists	IASTBinaryExpression	Main not found	MathsatInterpolatingProver	Unary	Aborted	Main/Label
Task description CPAchecker 06 Jun	2767	72	1	839	104	23	2	-	-
Task description CPAchecker 08 Jun	2767	330	-	6	-	-	2	-	-

Рис. 5.: Идентификация критичных ошибок в инструменте верификации CPAchecker



Total changes	Safe → Unsafe	Safe → Unknown	Unsafe → Unknown	Unknown → Safe	Unknown → Unsafe
657	3	537	32	80	5

Рис. 8.: Сравнение инструмента BLAST с CPAchecker (показаны изменения вердиктов BLAST→ CPAchecker)

медленней (35 часов), и выдает меньше вердиктов UNSAFE. Сравнение (рис. 8) показало, что имеется три дравера с вердиктом SAFE инструмента BLAST, для которых CPAchecker выдал вердикт UNSAFE. Анализ детального сравнения результатов показал, что в одном драйвере BLAST правильно доказал, что связный список пуст, чего не смог доказать CPAchecker и дошел до ошибочной ветки выдав UNSAFE. В двух драйверах BLAST неправильно вычислил константное выражение и поэтому не обнаружил ошибочный путь. В 32 драйверах CPAchecker выдал UNKNOWN вердикт, тогда как BLAST выдал UNSAFE. Детальный анализ показал, что почти половина UNKNOWN вердиктов инструмента CPAchecker связана с нехваткой памяти, а вторая половина – с ошибками синтаксического анализатора. В пяти UNSAFE, обнаруженных инструментом CPAchecker, для которых BLAST выдал UNKNOWN, какой-либо закономерности выявить не удалось.

5.5. *Подготовка данных для соревнований инструментов верификации*

В сентябре этого года стартовало первое международное соревнование инструментов верификации языка Си “Competition on Software Verification” (SV-COMP) [36], результаты которого будут объявлены на конференции TACAS 2012.

Специально для соревнования был осуществлен запуск одного из новых ядер ОС Linux на двух правилах, связанных с корректной работой захвата и освобождения модуля ядра и выделения и освобождения ресурсов подсистемы USB. По результатам этого прогона был выбран набор из наиболее сложных драйверов, которые были предоставлены организаторам соревнования (набор *ldv-drivers*). Кроме того, были предоставлены тесты из регрессионного тестового набора системы *LDV* (набор *ldv-regression*), включающего модульные тесты, нацеленные на проблемные места отдельных компонентов системы *LDV*. Данные наборы составили примерно третью часть от всех тестов соревнования SV-COMP (87 из 280).

Отметим, что для соревнований важно, чтобы имелся источник новых данных, на которых можно было бы сравнивать инструменты верификации так, чтобы они не подстраивались под конкретный тестовый набор. Важно, чтобы инструменты верификации на соревновании умели обрабатывать произвольные корректные конструкции языка Си, а не только те, которые встречаются в эталонном наборе для сравнения. Возможность автоматической генерации новых наборов данных из непрерывно развивающегося ядра ОС Linux позволит поставлять на соревнования новые тестовые данные по мере необходимости.

6. Заключение

Статья представляет архитектуру открытой системы верификации драйверов ОС Linux.

Система *LDV* уже показала себя пригодной для проверки качества драйверов, поставляемых вместе с ядром ОС Linux. С ее помощью было найдено более 40 ошибок в последних версиях ядра, которые были признаны сообществом разработчиков ядра [37].

Помимо этого, система верификации *LDV* является удобным инструментом для проведения экспериментов с различными методами статического анализа. Для этого система предоставляет следующие возможности:

- Определяет простой и явный интерфейс для подключения новых инструментов верификации и их запуска в различных конфигурациях.
- Дает возможность применять инструменты верификации к огромному массиву промышленного исходного кода на языке Си, которым являются драйверы устройств ОС Linux.
- Предоставляет удобный интерфейс визуализации для сравнения различных инструментов верификации, а также сравнения различных конфигураций одного инструмента.
- Позволяет выделять наборы тестовых данных для сравнения инструментов верификации.

Система *LDV* является свободным программным обеспечением, ее описание и исходный код доступны на сайте проекта *http://linuxtesting.org/project/ldv*.

СПИСОК ЛИТЕРАТУРЫ

1. *Engler D., Chelf B., Chou A.* Checking system rules using system-specific, programmer-written compiler extensions // *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4.* — 2000. — Pp. 1–16. `http://dl.acm.org/citation.cfm?id=1251229.1251230`.
2. *Сыромятников С.* Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык kast // *Труды Института системного программирования РАН.* — 2011. — Т. 20. — С. 51–68.
3. *Несов В., Маликов О.* Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ // *Труды Института системного программирования РАН.* — 2006. — Т. 9. — С. 51–56.
4. *Несов В., Гайсарян С.* Автоматическое обнаружение дефектов в исходном коде программ // *Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции.* — 2008. — С. 107.
5. *Nesov V.* Automatically finding bugs in open source programs // Third International Workshop on Foundations and Techniques for Open Source Software Certification. — Vol. 20 of *OpenCert 2009.* — 2009. — Pp. 19–29.
6. *Dillig I., Dillig T., Aiken A.* Sound, complete and scalable path-sensitive analysis // *SIGPLAN Not.* — 2008. — June. — Vol. 43. — Pp. 270–280. `http://doi.acm.org/10.1145/1379022.1375615`.
7. *Hovemeyer D., Pugh W.* Finding bugs is easy // *SIGPLAN Not.* — 2004. — December. — Vol. 39. — Pp. 92–106. `http://doi.acm.org/10.1145/1052883.1052895`.
8. *Evans D., Larochelle D.* Improving security using extensible lightweight static analysis // *IEEE Softw.* — 2002. — January. — Vol. 19. — Pp. 42–51. `http://portal.acm.org/citation.cfm?id=624647.626359`.
9. The software model checker Blast: Applications to software engineering / D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar // *Int. J. Softw. Tools Technol. Transf.* — 2007. — Vol. 9, no. 5. — Pp. 505–525.
10. *Shved P., Mutilin V., Mandrykin M.* Static verfication "under the hood": Implementation details and improvements of BLAST // Proceedings of SYRCoSE. — Vol. 1. — 2011. — Pp. 54–60.
11. *Beyer D., Keremoglu M. E.* CPAchecker: a tool for configurable software verification // Proceedings of the 23rd international conference

on Computer aided verification. — CAV'11. — Berlin, Heidelberg: Springer-Verlag, 2011. — Pp. 184–190. http://dl.acm.org/citation.cfm?id=2032305.2032321.

12. *Clarke E., Kroening D., Lerda F.* A tool for checking ANSI-C programs // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) / Ed. by K. Jensen, A. Podelski. — Vol. 2988 of *Lecture Notes in Computer Science*. — Springer, 2004. — Pp. 168–176.

13. *Podelski A., Rybalchenko A.* ARMC: The logical choice for software model checking with abstraction refinement // Practical Aspects of Declarative Languages / Ed. by M. Hanus. — Springer Berlin / Heidelberg, 2007. — Vol. 4354 of *Lecture Notes in Computer Science*. — Pp. 245–259.

14. The static driver verifier research platform / T. Ball, E. Bounimova, V. Levin et al. // Computer Aided Verification. — CAV'10. — 2010. — Pp. 119–122.

15. SLAM2: Static driver verification with under 4% false alarms / T. Ball, E. Bounimova, R. Kumar, V. Levin // Formal Methods in Computer Aided Design. — 2010.

16. *Khoroshilov A., Mutilin V.* Formal methods for open source components certification // 2nd International Workshop on Foundations and Techniques for Open Source Software Certification. — OpenCert 2008. — 2008. — Pp. 52–63.

17. How to cook an automated system for Linux driver verification / A. Khoroshilov, V. Mutilin, V. Shcherbina et al. // 2nd Spring Young Researchers' Colloquium on Software Engineering. — Vol. 2 of *SYRCoSE 2008*. — 2008. — Pp. 11–14.

18. Establishing Linux driver verification process / A. Khoroshilov, V. Mutilin, A. Petrenko, V. Zakharov // Perspectives of Systems Informatics / Ed. by A. Pnueli, I. Virbitskaite, A. Voronkov. — Springer Berlin / Heidelberg, 2010. — Vol. 5947 of *Lecture Notes in Computer Science*. — Pp. 165–176.

19. Towards an open framework for C verification tools benchmarking / A. Khoroshilov, V. Mutilin, E. Novikov et al. // Proceedings of PSI. — 2011. — Pp. 82–91.

20. Архитектура Linux driver verification / В. Мутилин, Е. Новиков, А. Страх и др. // *Труды Института системного программирования РАН*. — 2011. — Т. 20. — С. 163–187.

21. An empirical study of operating systems errors / A. Chou, J. Yang, B. Chelf et al. // SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles. — New York, NY, USA: ACM, 2001. — Pp. 73–88.

22. *Swift M. M., Bershad B. N., Levy H. M.* Improving the reliability of commodity operating systems // SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles. — New York, NY, USA: ACM, 2003. — Pp. 207–222.

23. *В.П.Иванников, Петренко А.* Задачи верификации ОС Linux в контексте ее использования в государственном секторе // *Труды Института системного программирования РАН*. — 2006. — Т. 10. — С. 9–14.

24. *Kroah-Hartman G., Corbet J., McPherson A.* Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. — http://www.linuxfoundation.org/docs/lf_linux_kernel_development_2010.pdf. — 2010.

25. *Ball T., Rajamani S. K.* SLIC: A specification language for interface checking: Tech. rep.: Microsoft Research, 2001. http://research.microsoft.com/apps/pubs/default.aspx?id=69906.

26. Proofs from tests / N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons // Proceedings of the 2008 international symposium on Software testing and analysis. — ISSTA '08. — New York, NY, USA: ACM, 2008. — Pp. 3–14. http://doi.acm.org/10.1145/1390630.1390634.

27. Model checking concurrent Linux device drivers / T. Witkowski, N. Blanc, D. Kroen-

ing, G. Weissenbacher // ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. — New York, NY, USA: ACM, 2007. — Pp. 501–504.

28. *Post H., Küchlin W.* Integrated static analysis for Linux device driver verification // Proceedings of the 6th international conference on Integrated formal methods. — IFM'07. — Berlin, Heidelberg: Springer-Verlag, 2007. — Pp. 518–537. http://portal.acm.org/citation.cfm?id=1770498.1770525.

29. SATABS: SAT-based predicate abstraction for ANSI-C / E. Clarke, D. Kroening, N. Sharygina, K. Yorav // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005). — Vol. 3440 of *Lecture Notes in Computer Science*. — Springer Verlag, 2005. — Pp. 570–574.

30. *Post H., Küchlin W.* Automatic data environment construction for static device drivers analysis // Proceedings of the 2006 conference on Specification and verification of component-based systems. — SAVCBS '06. — New York, NY, USA: ACM, 2006. — Pp. 89–92. http://doi.acm.org/10.1145/1181195.1181215.

31. CIL: Intermediate language and tools for analysis and transformation of C programs / G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer // Proceedings of the 11th International Conference on Compiler Construction. — CC '02. — London, UK: Springer-Verlag, 2002. — Pp. 213–228. http://portal.acm.org/citation.cfm?id=647478.727796.

32. *Новиков Е.* Упрощение анализа трасс ошибок инструментов статического анализа кода // *Труды второй научно-практической конференции «Актуальные проблемы системной и программной инженерии» (АПСПИ-2011).* — 2011. — 25 Мая. — С. 215–221.

33. Incremental benchmarks for software verification tools and techniques / B. W. Weide, M. Sitaraman, H. K. Harton et al. // Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments. — VSTTE '08. — Berlin, Heidelberg: Springer-Verlag, 2008. — Pp. 84–98. http://dx.doi.org/10.1007/978-3-540-87873-5_10.

34. Counterexample-guided abstraction refinement / E. Clarke, O. Grumberg, S. Jha et al. // *Proc. CAV, LNCS.* — 2000. — Vol. 1855. — P. 154–169.

35. *Beyer D., Henzinger T. A., Théoduloz G.* Configurable software verification: concretizing the convergence of model checking and program analysis // Proceedings of CAV. — Berlin, Heidelberg: Springer-Verlag, 2007. — Pp. 504–518. http://portal.acm.org/citation.cfm?id=1770351.1770419.

36. *Web-site.* 1st International competition on software verification (SV-COMP) held at TACAS 2012. — http://sv-comp.sosy-lab.org.

37. *Web-site.* Problems found in Linux kernels. — http://linuxtesting.org/results/ldv.